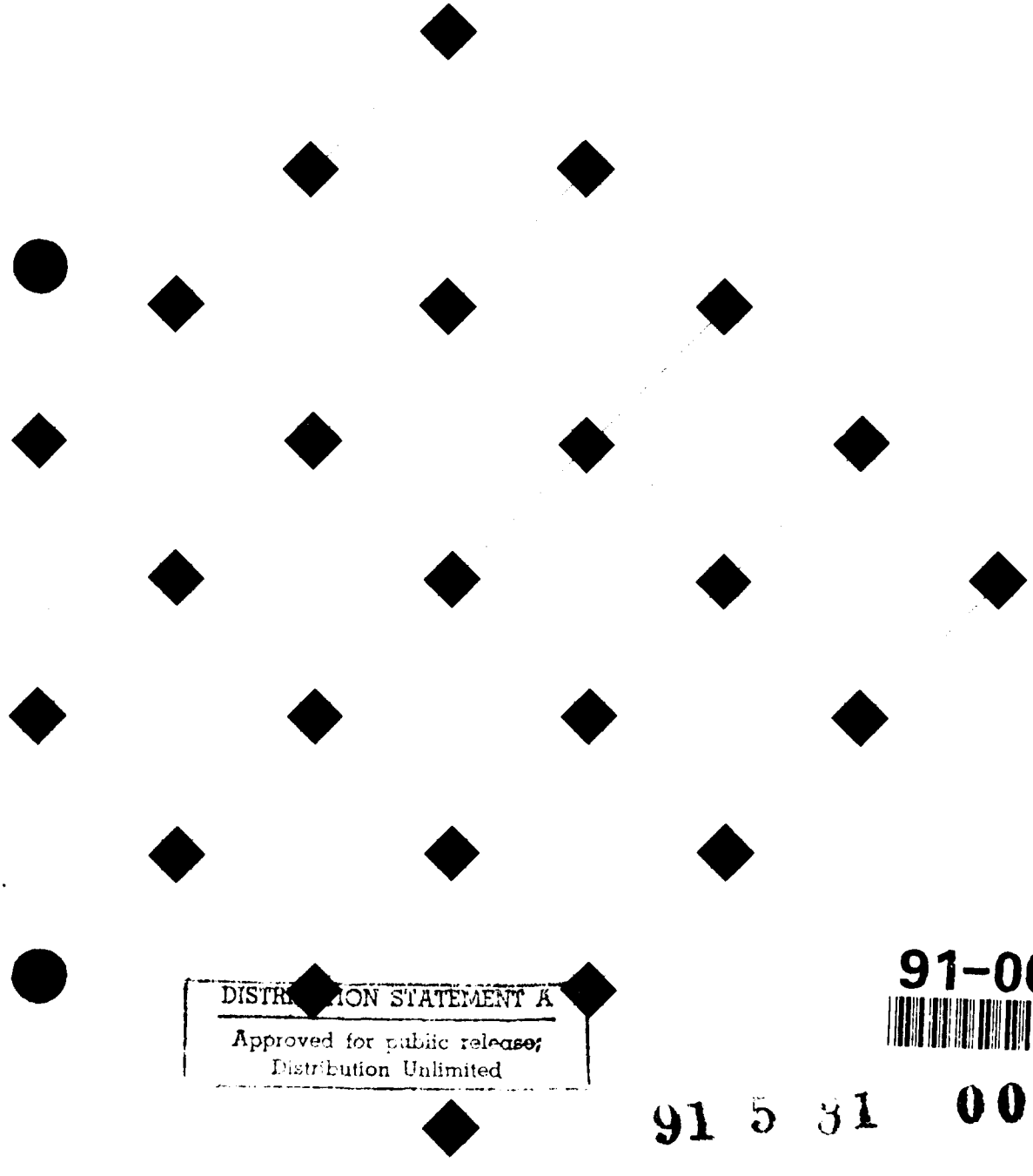
Carnegie Mellon University
Software Engineering Institute

Introduction to Software Design

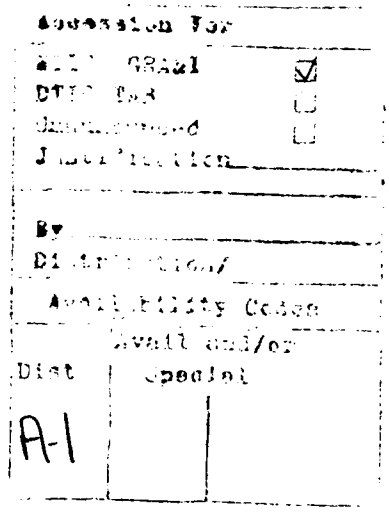
Curriculum Module SEI-CM-2-2.1



Introduction to Software Design

SEI Curriculum Module SEI-CM-2-2.1

January 1989



David Budgen
University of Stirling

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

Draft For Public Review

This technical report was prepared for the

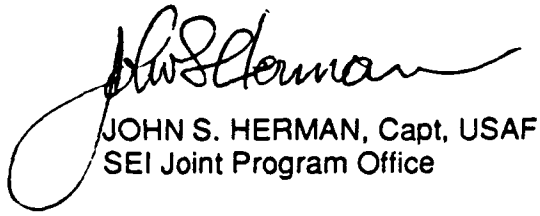
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Foreword

In 1986, David Budgen was one of the original SEI curriculum module authors. He, with co-author Richard Sincovec, wrote the first version of this module and, in so doing, helped define what a module should be. In the two years since Professor Budgen completed his work, the module concept has evolved, and the detailed modules on design originally envisaged to complement that early module have not materialized. A workshop was convened at the SEI in the spring of 1988 to revisit CM-2 and to devise a plan for better serving the needs of software engineering educators for material on software design.[1] Workshop participants concluded that the original module provided a good basis for a more ambitious introduction and made numerous suggestions concerning the content and organization of a successor. Fortunately, Professor Budgen was able to return to the SEI in the summer of 1988 to make such a revision—really a rewrite—of his module. To do this, the author drew upon his recent experience teaching design, reviewer comments, the recommendations from the design workshop, and the work and methods of other module authors. Although minor revisions remain to be made, this new version of *Introduction to Software Design* should provide a helpful, insightful commentary on an important software engineering topic.

— Lionel E. Deimel
Senior Computer Scientist, SEI

[1] Participants in the workshop were David Card (Computer Sciences Corp.), Raymonde Guindon (MCC), Everett Merritt (IBM), Richard Sincovec (University of Colorado), David Weiss (SPC), and, from the SEI, Mark Ardis, Lionel Deimel, David Glass, and John Nestor.

Introduction to Software Design

Acknowledgements

The second version of this module has benefited from the contributions of numerous people, including many who assisted in the production of the first version. I would particularly like to acknowledge the contributions of John Nestor and Jan Pedersen, who helped resolve some of the major structural issues that surfaced in the revision process, and the comments and ideas of Glenn Bruns and Hassan Gomaa. Despite all these valiant efforts, there are, no doubt, some remaining errors or ambiguities, for which I must take sole responsibility.

The original version of this module was written with Richard Sincovec, and we would like to thank Paul Jorgensen, Glenn Bruns, and David Weiss, all of whom made major contributions. We also acknowledge the helpful comments of Larry Peters, Michael Jackson, and Bob Glass.

Contents

Introduction to Software Design

Module Revision History

Version 2.1 (January 1989)	Editorial corrections by author
Version 2.0 (December 1988)	Major revision of structure and content
Version 1.2 (July 1987)	Format changes for title page and front matter; removed references to worked examples that are not yet available; added acknowledgements
Version 1.1 (April 1987)	Slight cosmetic changes
Version 1.0 (September 1986)	Original version

Introduction to Software Design

Capsule Description

This curriculum module provides an introduction to the principles and concepts relevant to the design of large programs and systems. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

Philosophy

Design is an important activity for all except the most trivial of systems. It exerts a major influence upon the other phases of the development process, as well as upon system maintenance. An understanding of *design issues* and of the techniques available to assist in producing a design is essential background for the software engineer.

This module provides an introduction to the topic of *software design*, including the following major elements:

- An explanation of the *role* of the design activity in producing large software-based systems, together with an introduction to the principles that are used to assess the *quality* of a design.
- An introduction to a range of *design representations*, together with a description of their uses and limitations.
- An explanation of the *role* of a design method in the production of a design and of the design *strategies* used in software design methods.
- An introduction to several examples of design methods and an *assessment* of their strengths and limitations with respect to different classes of problems.

The module places strong emphasis upon providing an understanding of design as a general problem-solving activity and upon how it differs from the problem-solving techniques of more established disciplines. Within this context, it is possible to understand the role of design methods and the limitations inherent within them.

The design of software is essentially a creative operation, but the designer of a large system usually requires at least guidelines, and preferably a *method*, to provide a structure for this task. The state of the art is such that this module cannot advocate one design approach alone. Its emphasis, therefore, is on identifying *principles*. In order to illustrate these, a number of methods have been selected as representative examples of the different design strategies in use and of the current best practice in software design. Since the module is an introduction to the topic of design, the design methods chosen are only described in outline, to a level considered sufficient to illustrate the principles involved. No attempt has been made to explain every nuance and special case that might arise when using any particular method.

The module contains material needed for a basic understanding of the design process (and therefore as a prerequisite for any advanced study of design). These topics might be taught in a software design course or as part of an introductory course on software engineering. Since an understanding of the issues and trade-offs that arise in *system design* requires an understanding of the structuring of software systems, the material in this module should also be considered as a necessary prerequisite to any study of system design.

In writing this curriculum module, an effort has been made to conform to the general framework for describing software development processes and products introduced by H. Dieter Rombach in his curriculum module *Software Specification: A Framework* [Rombach87]. Professor Rombach's module is a useful prerequisite for understanding the terminology used here. Figure 1 shows the relationships

among processes and products closely related to software design, using the nomenclature of [Rombach87]. Figure 2 shows a simplified representation of the design process, omitting the inevitable iterative details. This process is concerned with how a system can be built so as to behave in the manner described by the D-requirements product. During the design process, further documents are generated, which in turn provide inputs both to the detailed design and to implementation tasks. The output of the design process is a design product, which is the input to software implementation.

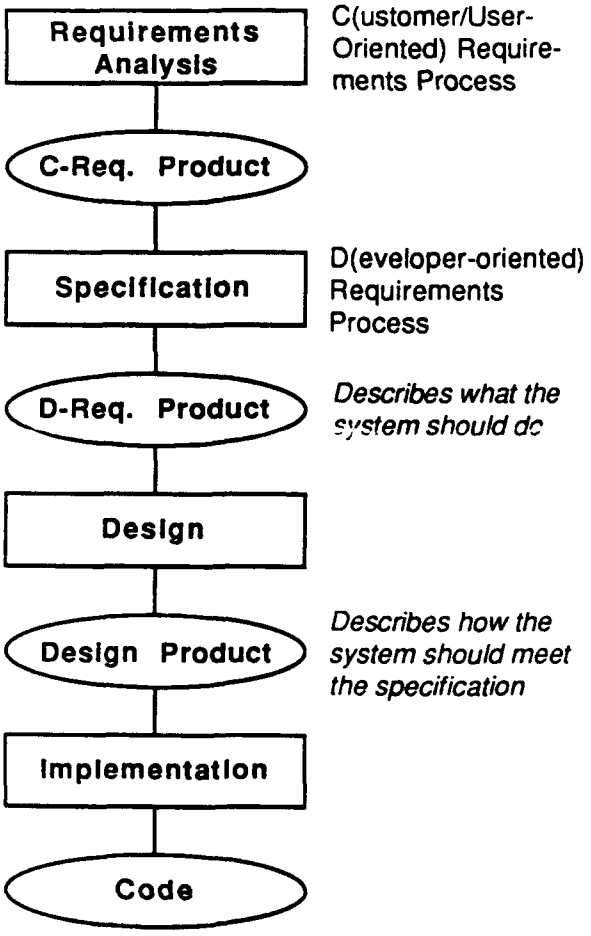


Figure 1. Process-product relationships.

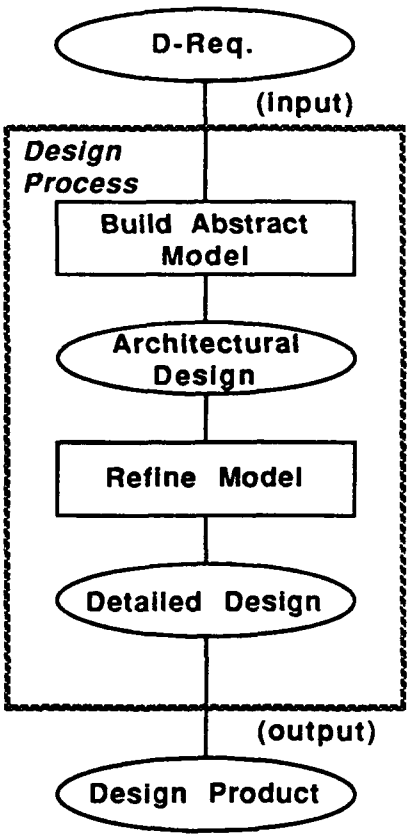


Figure 2. Design process.

Objectives

The student who has worked through a complete selection of material from this module is expected to:

- Be able to *explain* the role of design in the production of software systems and understand the use of *abstraction* in the design process.
- Be *aware* of the differences between designing as an individual and as part of a team, and of the need to *record* the process of decision-making during design.
- Be familiar with the practices and representation forms used in those *design methods* that represent current best practice, and be able to explain the principles behind each, their principal domains of application, and their limitations (this can be summarized as *assessment of the process*).

- Be able to *identify* methods of assessing the results of the design process in terms of the quality of a design, the extent to which it meets the original requirements, and its likely performance, where appropriate (this can be summarized as *assessment of the product*).
- Have been *involved* in a number of exercises that are intended to amplify and illustrate the points made above.

The material of this module is not intended to be sufficient to provide the student with a comprehensive understanding of any particular design method. Rather, he will be able to gain an understanding of the domains of application that are best suited to each method and an appreciation of the trade-offs that occur during the design process. The student should then be equipped to assist a more experienced designer, as the next stage of his learning process, and should have acquired the background that is needed for reading some of the more specialized texts on particular design methods.

Prerequisite Knowledge

The student should already have acquired a fairly thorough knowledge of the techniques of *programming-in-the-small* and of the use of *abstraction*, although this module is not directly concerned with implementation. It is necessary for the student to possess a degree of programming experience in order to be able to understand the reasons behind some of the design decisions that need to be made, even where these are at a fairly abstract level. For example, a student undertaking a course based upon this module typically should have written and modified programs of at least 500 lines of source code and should be familiar with such data structures as stacks, linked lists, queues, and trees.

While this module is concerned with both sequential and parallel-processing problems, the student need only be familiar with the basic *concepts* of parallel processing. That is, the student should understand the concepts of *synchronization* and *mutual exclusion*, although he need not be familiar with the usual details of their implementation.

The material of the module requires no specific knowledge of computer hardware, although the student is expected to have some basic knowledge of conventional von Neumann architectures.

Module Content

A glossary of important terms follows the annotated outline.

Outline

I. The Role of Software Design
  1. The Design Process
    a. Definition of design
    b. Objectives of the design process
    c. Design as a problem-solving process
    d. Design as a "wicked" problem
    e. Design as a model-building process
    f. The role of a design method
    g. Constraints on the design process
    h. Recording the process of design
    i. Design by an individual vs. design by a group
  2. Design as a Step in System Development
    a. Relationship of design to other activities
    b. Production models
    c. Economic factors
    d. Roles of prototyping
  3. Principles of Design
    a. Abstraction
    b. Modularity
    c. Information-hiding
    d. Completeness
    e. Design for maintenance
    f. Design for reuse
    g. Assessing a design
    h. Design verification

II. Design Practices: General Issues
  1. Role of Design Methods
    a. Reasons for using a design method
    b. Management benefits arising from the use of a method
    c. Limitations of design methods
    d. General forms of systematic and formal design methods
  2. Classification of Systems
    a. Batch systems
    b. Reactive systems
    c. Concurrent systems
  3. Design Strategies
    a. Top-down strategies
    b. Design by composition, evolution of design methods
    c. Stylized design
  4. Design Representations
    a. Data flow diagrams (DFD)
    b. HIPO diagrams
    c. Structure charts
    d. Decision tables
    e. Entity structure diagrams (JSD)
    f. System specification diagrams (JSD)
    g. Entity history diagrams
    h. Structure graphs
    i. Finite state machines
    j. Statecharts
    k. Petri nets
    l. Pseudocode
    m. Formal design languages

III. Design Practices: Design Methods
  1. Structured Systems Analysis and Structured Design
    a. Problem decomposition (SSA)
    b. Create a data dictionary
    c. Describe process logic
    d. Deriving a structured design from the logical model
    e. An assessment of SSA/SD
  2. Design by Modeling the Problem: JSP and JSD
    a. JSP and JSD principles and the relationship between the two methods
    b. JSP design process
    c. Handling multiple inputs in JSP—structure clashes
    d. Program inversion
    e. An assessment of JSP
    f. Extension of philosophy to JSD

g. Concept of entity-action-attribute
h. Steps involved in JSD
i. An assessment of JSD

3. Object-Oriented Design (OOD)
   a. The concept of *objects*
   b. Abstraction, information-hiding, modularity and localization
   c. Steps involved in object-oriented design
   d. Limitations/problems in applying object-oriented design
   e. An assessment of OOD
4. Some Other Systematic Design Methods
   a. Structured Analysis and Design Technique (SADT)
   b. The Warnier-Orr approach
   c. Stepwise Refinement with Verification (Mills)
   d. SSADM

IV. Review of Design Practices
1. Some Assessments of Design Methods
   a. General assessment of methods
   b. The use of software tools to support design methods
   c. Selecting appropriate design approaches for classes of problems
2. Trends and Developments
   a. Evolution of design methods
   b. Trends in the development of design methods

Annotated Outline

I. The Role of Software Design

The purpose of this first section is to identify the role and objectives of the design process, the context within which it takes place, and the products that should result from it. The material provided is intended to achieve this purpose, rather than to describe how design is actually performed.

We can also identify some of the *constraints* and *limitations* that apply to the design process generally, both in terms of the techniques currently available and of our ability to control and manage the process. This information provides much of the background required for understanding the form of the design process.

1. The Design Process

Design is a process carried on in many spheres of human activity. It typically involves the designer in drawing upon his experience, together with a degree of creative ability, in order to formulate and evaluate a solution for a given problem. Here, we are concerned with the process of designing software systems. The design process within other disciplines, and particularly in other branches of engineering, may usefully be compared and contrasted with the design process conducted for software development.

A more general view of the design process in a wider context is provided in [Jones70], and it is worth quoting his explanation of "why design is difficult" at the outset:

> *The fundamental problem is that designers are obliged to use current information to predict a future state that will not come about unless their predictions are correct. The final outcome of designing has to be assumed before the means of achieving it can be explored: the designers have to work backwards in time from an assumed effect upon the world to the beginning of a chain of events that will bring the effect about.*

These factors apply as much to software design as to any other form of design, and they are further augmented by the fact that we are almost always concerned with the production of systems for new and original applications of computers.

a. Definition of design

The purpose of design is to specify a solution to a given problem. (Usually this problem is expressed as a functional specification.) The designer postulates a solution, models it, evaluates it against the original requirements, and, after some iteration of these operations, produces a detailed specification of the solution for the programmer to implement. (In this context, the functional specification is regarded as being equivalent to the "D-requirements" identified in [Rombach-87].)

b. Objectives of the design process

The objective of the design process is to produce a set of detailed specifications that describe the intended form of implementation for the software system. These specifications describe both the form (structure) of the solution and the way that the components are to fit together, and so act as a set of "blueprints" that show how the system is to be constructed.

c. Design as a problem-solving process

Design can be regarded as a form of problem-solving process that involves making extensive use of abstraction, including separation of the logical aspects of the design from the physical

aspects of the design. Design involves making choices, often involving tradeoffs between the different qualities the designer is seeking to achieve in his solution. The ultimate criterion must be that of "fitness for purpose," in that the solution should not only exhibit the best possible structure, but must also do the required job as well as possible.

d. Design as a "wicked" problem

A "wicked" problem has been described as a form of problem where the solution to one of its aspects may reveal an even more serious difficulty. Software design can be considered an example of this type of problem. See [Peters81] for an interesting discussion of this issue.

e. Design as a model-building process

The process of design involves the designer in first building a highly abstract model of the chosen solution—which the designer may possibly "execute" symbolically—and then translating this into a detailed structure to act as the blueprint for construction. It is generally possible to distinguish between "architectural design" and "detailed design" phases. The former is concerned with the general structure of a solution. It may be influenced by consideration of the effects of factors such as the choice between implementation using distributed or single processors, the need for compatibility with existing structures, and likely future developments. Detailed design is more concerned with the formulation of blueprints for the particular solution and with modeling the detailed interaction between its components. For a discussion of the importance of model-building in software design, see [Adelson85].

f. The role of a design method

The role of a *design method* is to provide assistance with the model-building and with the translation process. A design method can be viewed as a plan of action based on a set of decision-making criteria (the "process" part) and supported by diagrammatic or symbolic forms that aid in building a particular form of model (the "representational" part). The representations may also provide a framework that assists with evaluating the consequences of making particular design choices. It may be necessary to make some top-level decisions about the overall architecture of a system before applying any method, and some of these decisions will then act as *constraints* upon the form of the final design.

g. Constraints on the design process

Ideally, the designer is concerned solely with producing the "best" possible solution to a problem. In practice, there will almost always be a set of constraints restricting architectural and other characteristics of the solution produced. Company design practices, standard hardware configurations, existing file structures, real-time constraints, implementation language features, and the need to anticipate future changes all restrict the solution space available to the designer. It is important to appreciate that we rarely design anything in total isolation and that some requirements for compatibility nearly always exist.

h. Recording the process of design

It is important to record the history of the evolution of a design, particularly, the *reasons* for making specific choices. This helps both with design audits and with the maintenance task, since the maintenance designer needs to know why particular choices were made and why other options were discarded. Unfortunately there is no consensus about how this recording should be done. Current design methods do not include it as part of the design process, nor do they provide any specific forms of representation for the purpose. Such recording as does occur is likely to be either by an individual or according to some (relatively arbitrary) company practice.

i. Design by an individual vs. design by a group

The process of design is made more complex when more than one designer is involved, since the designers need to find a way of "splitting" the problem. A group effort requires that an additional process of "negotiation" occur when the components of the design are brought together. A good decomposition of design tasks is one that provides minimal, well-structured interfaces between modules designed by different people.

References:

Papers: Adelson85

Books: Abbott86, Birrell85, Fairley85, Jones70, Peters81, Pressman82, Rombach87

2. Design as a Step in System Development

Design is a major phase in the development of software systems. The forms and roles it takes on in two widely-used production models are described here. The prototyping model is especially useful in the design of interactive systems and expert systems.

a. Relationship of design to other activities

The role of design is best understood in the context of the other activities involved in producing a plan of action for the implementation. **Requirements analysis** identifies what is *needed* in a system; **specification** describes what the system

should *do*; and **design** describes *how* that should be done. Figure 1 shows these relationships and the resulting products. See [Rombach87] for a fuller description of these roles.

b. Production models

While such tasks as specification, design, and implementation need to be performed in all cases, there are a number of ways in which their interactions can be organized. The two main forms are described below.

(i) Waterfall model

This is widely discussed in the literature, although the detailed form shows some variability. See the descriptions in [Birrell85] and in [Fairley85] for clear expositions, with diagrams.

(ii) Incremental enhancement model

This is one role for the use of *prototyping* (see below) and is discussed in the same references cited for the waterfall model.

c. Economic factors

It is important to detect and eliminate errors during design, since the earlier in the development process that errors are detected, the cheaper it should be to correct them. The cost of software maintenance is also affected significantly by the quality of a design. The economic issues of software production and maintenance are discussed in [Boehm81].

d. Roles of prototyping

For a fuller discussion of the different ways in which prototyping may be used for the development of software-based systems, see the paper by Floyd in [Budde84]. The three main categories of use identified by Floyd are:

- *Evolutionary*—adapting the system gradually to changing requirements, which cannot be determined reliably in the earliest stages of development (incremental development). The prototype eventually becomes the product.
- *Experimental*—used to determine the adequacy of a proposed solution before investing in large-scale implementation. This may involve investigating such features as performance and resource needs, as well as the details of the human-computer interface. The prototype is essentially "throwaway" code.
- *Exploratory*—used to clarify requirements and desirable features of the target system, and to evaluate alternate solutions. The prototype should be "throwaway" code, and can be considered as enhancing the requirements analysis and functional specification information.

A rather different approach to the use of prototyping techniques, with an emphasis on the use of executable specifications and a more formal approach to design, is discussed in [Henderson86].

References:

Papers: Belady76, Henderson86
Books: Birrell85, Boehm81, Budde84, Fairley85, Pressman82, Rombach87, Sommerville85

3. Principles of Design

It is possible to identify certain properties that we expect a good design to possess in some measure, with the balance of emphasis being apportioned among these according to the class of problem and the needs of the particular application. Identifiable features that a design should exhibit include those related to the functioning of the system, such as:

- *Fitness for purpose*—the system must work, and work correctly, in that it should perform the required tasks in the specified manner and within the constraints of the specified resources.
- *Robustness*—the design should be stable against changes to such features as file and data structures, user interface, etc.

Desirable features include those facilitating maintenance and reuse, including:

- *Simplicity*—the design should be as simple as possible, but no simpler.
- *Separation of concerns*—the different concepts and components should be separated out (related closely to *modularity*).
- *Information hiding*—information about the detailed form of such objects as data structures and device interfaces should be kept local to a module or unit and should not be directly "visible" outside that unit.

We can also identify some features that a bad design is likely to exhibit and that will make it difficult to read and understand the designer's intentions. Among these are:

- Having too much retained state information spread around the system.
- Using interfaces that are too complex.
- Containing excessively complex control structures.
- Using modules lacking functional strength.
- Involving needless replication.

The issues that form the topics of this section are all

related to these lists of features. Because such features are generally only made manifest through consideration of relatively detailed design structures, it is also desirable to be able to relate them to more abstract concepts, and so to the use of abstraction in design. As a further categorization, it is useful to distinguish between the *constructional* issues (which are essentially concerned with packaging and dependency), and the *runtime* issues (which involve making decisions about such features as concurrency and the calling hierarchy of procedures).

a. Abstraction

The increasing use of abstraction has been one of the major factors in the development of a more structured approach to software design. The use of abstract objects and operations upon them needs to be seen as central to any attempt to produce a well-engineered design. Abstraction allows the designer to model *logical* structures as well as *physical* structures (or *properties* as well as *representations*).

b. Modularity

This can be related to ideas about such issues as *separation of concerns* and *simplicity*. Two well-established measures for assessing the partitioning of a system into "modules" (which may be implemented as programs, packages, subprograms, etc.) are *cohesion* and *coupling*. *Cohesion* is concerned with the relationships among the elements making up a module, while *coupling* is concerned with the interdependencies between different modules. To make practical use of these, the different forms that each measure can take must be related to particular implementation issues, such as the use of global variables. A good description of these measures can be found in [Page-Jones80].

(i) Forms of cohesion

Seven forms of module cohesion (sometimes termed *association*) are commonly recognized. In order, from highly desirable to undesirable these are:

1. functional,
2. sequential,
3. communicational,
4. procedural,
5. temporal,
6. logical, and
7. coincidental.

(ii) Forms of coupling

While coupling is more quantifiable than cohesion, the terminology used for the descriptions of the main forms may vary a little more. A list of forms, roughly ranked from desirable to undesirable, is:

1. data coupling,
2. stamp coupling,
3. control coupling,
4. common-environment coupling, and
5. content coupling.

c. Information-hiding

This principle is widely accepted as a design criterion [Parnas72, Parnas79] and forms a basis for assessing a choice of modular structure. Basically, it involves concealing the details of the structure and forms of certain objects and ensuring that these can only be accessed by those procedures provided to implement the operations on those abstract objects.

d. Completeness

This is primarily an issue of whether the design meets all of the requirements of the specification, including any real-time or similar operational constraints. It is largely concerned with the concept of *fitness for purpose*.

e. Design for maintenance

Since the maintenance of software usually absorbs greater effort than its production, the design process should recognize the need for future changes and modifications/enhancements of a system, as indeed is encouraged by considering the use of information hiding. Consideration of possible future changes also emphasizes the need for a design to be *robust* against possible changes. This issue generally supports the need to use good practices, rather than imposing specific needs, but it may impose requirements upon the detailed form of the design, too. See [Birrell85] for a good discussion of these points.

f. Design for reuse

Reuse represents a rather ill-defined and poorly understood area. While the practice of reuse of software components is long-established and can be extended to include the use of *generic* forms, the reuse of design in any form is still relatively new in concept. Ways of organizing for the reuse of designer experience is quite well-established in other branches of engineering, but within software engineering even the reuse of a designer's own experience remains highly domain-dependent [Adelson85]. Reuse is an issue of concern when considering design methods in detail.

g. Assessing a design

Design assessment is concerned with two major issues:

- how well the design meets the specification (*completeness* and *correctness*) and
- how well-structured the design is (*quality*).

In addition, any assessment must consider both

- the *static* structures of the design and
- the *dynamic* performance of the implemented system.

In order to match a design against the specification, we are able to make use of design reviews and expert opinion (see [Yourdon85]). In order to assess the dynamic aspects, we may also be able to use some form of operational model or prototype. Where formal methods are used for both specification and design, it may also be possible to use mathematical techniques for assessing the static issues.

The assessment of *quality* is more subjective. Although some of the same techniques can be used (such as design reviews), there is a need for some form of design *metrics* to support these. Unfortunately metrics can only be used effectively with well-defined design representations, and so there are relatively few guidelines available. (See [Mills88] for a fuller description of metrics used with software.) Again, any assessment of the dynamic qualities will need to make use of some form of modeling technique.

h. Design verification

Formal methods make use of mathematical techniques to provide a means of verifying an initial design for completeness and consistency. They supply the formal correspondence between specification and design [Berztiss87], so that the design can be verified for completeness and consistency against the specification. Two of the most widely used approaches are those based upon algebraic forms (such as OBJ-2, [Futatsugi85]) and the model-oriented forms (such as VDM [Bjørner82, Pedersen88] and Z [Hayes87]). Particular problem areas for these methods include handling concurrency and the extent of the mathematical knowledge needed for their use.

References:

Papers: Parnas72, Parnas79, Stevens74
Books: Abbott86, Berztiss87, Berztiss88, Birrell85, Bjørner82, Hayes87, Jensen79, Page-Jones80, Pedersen88, Yourdon79, Yourdon85

II. Design Practices: General Issues

The material of the next few sections is concerned with examining the main managerial and technical features of the design process. These sections are mainly concerned with establishing the rationale for using a design method, identifying the general forms that software design methods have, and explaining the reasons why no one method can be regarded as being capable of meeting the needs of other than a particular class of problems.

1. Role of Design Methods

For those unfamiliar with *programming-in-the-large*, some of the rationale for using a design practice (of any form) may need to be explained. The effects upon long-term factors such as maintenance should also be emphasised.

a. Reasons for using a design method

The reasons for using a design method need to be made explicit. A design method provides a *systematic* means of organizing and structuring the design process, as well as a set of criteria to assist in making choices. It can provide a checklist of actions, a means of building around particular principles, and may provide further assistance through the use of particular forms of notation or diagrams. The importance of such an approach is particularly significant for larger systems, where the use of a method establishes a common set of design goals for all the participants.

b. Management benefits arising from the use of a method

The management benefits of using one or more methods should be emphasised. The use of a "standard" method makes it easier for maintenance designers to model and assess the likely effects of any changes they might need to make, since it helps them to understand the ideas and models used by the original designer and to build an equivalent model for themselves. During system development, the use of a method also eases the hand-over of a design whenever staff changes occur, since the new designers need to be able to reconstruct the necessary models and understand the reasons behind particular choices.

c. Limitations of design methods

While a design method helps with the organization of the design process, it does not, in any sense, provide a "recipe" for producing a design for a particular problem. Each instance of the design process will be both domain- and application-specific, and takes place within the constraints that the context imposes. (A useful analogy is to think of a design method as being a set of instructions for producing a recipe. There are detailed directions for laying out the pages and producing the photographs, but little guidance as to how much seasoning to put into a particular dish.)

Design methods are also apt to be strongly oriented toward one domain of application, through the criteria they use and the weightings these criteria are given. The user of a method needs to be fully aware of any such underlying assumptions.

d. General forms of systematic and formal design methods

Design methods can be broadly classified as either *systematic* methods or *formal* methods. Formal methods largely depend upon the use of mathematical notation in order to allow consistency checking and rigorous transformations. Systematic methods are generally less mathematically rigorous in form, and usually consist of a "process" part describing what actions should be performed, and a "representational" part that describes how relevant structures may be represented. Some systematic methods, such as JSP or SSADM, may be highly *prescriptive* in nature, while others specify the actions for each step of the design process less completely. In general, the techniques from systematic methods can be combined, and can make use of representations adapted from other forms, when and as appropriate.

References:

Books: Bergland81, Birrell85, Fairley85

2. Classification of Systems

When describing problems that are to be "solved" through the use of software-based systems, we often group those that possess similar characteristics and refer to these as a "problem domain." Such a domain may be very broad (for example, "data-processing systems," "real-time systems") or quite tightly-defined (for example, "compilers"). For particular design methods, we also refer to the "domain of application," by which we generally mean the classes of problem for which a particular method is well-suited. Both the "problem-oriented" and "solution-oriented" views may be of use to a designer at different times.

Because the domains of application for design methods are ill-defined, it is generally impossible to attempt to make any "comparative methods" form of evaluation or assessment. However, some general scheme of classification of systems may help in discussing particular methods, and a scheme of batch, reactive, and concurrent forms is described below. Many problems will lead to software systems that are a combination of more than one of these classifications, so they should not be considered as being mutually exclusive. Real-time and embedded systems can generally be classified in this way, but they involve additional constraints upon size, performance, and structure.

a. Batch systems

The main feature of a batch system is that all of its operating characteristics are essentially determined when it begins processing one or more data streams. Any changes that occur to these characteristics arise because of the contents of the streams, when considered as sequential flows of data. Such a system should perform operations that are deterministic and repeatable. (An example of a batch system is a compiler.)

b. Reactive systems

The principal characteristic of a reactive system is that it is *event-driven*, the events being almost always asynchronous and non-deterministic. (A screen editor is an example of a reactive system.) In addition, the specifications of the required responses to events often include quite explicit requirements about timing.

c. Concurrent systems

Such systems are characterized by the use of multiple threads of execution, utilizing one or more processors. They generally require that the process of design should consider such issues as scheduling overhead, mutual exclusion, and synchronization of processes in the system.

References:

Papers: Bergland81

Books: Allworth87, Bergland81, Connor85

3. Design Strategies

This section aims to identify the general design strategies that underly the different methods discussed in the next few sections. At this level, we are only concerned with how these methods differ in terms of "strategy" and with the possible shortcomings or strengths of the various approaches.

There are two broad strategies that can be adopted. The first begins with a very abstract description of a solution and gradually refines this to produce a more detailed solution (*stepwise refinement*). This strategy is often referred to as a *top-down* process, and it is essentially requirements-based. The second strategy is based upon modeling the problem domain in some way, with the purpose of gradually building up a solution by adding features and viewpoints. This can be considered as a strategy based upon a process of *composition*.

a. Top-down strategies

The top-down approach can be considered a "divide and conquer" approach to problem-solving and design. The focus of the approach is functional decomposition. The need for a full understanding of the problem at the outset is essen-

tial, since most of the important decisions must be made early on in the design process. Wrong decisions may therefore lead to significant problems or result in the need for a major redesign. Even where decisions made are not necessarily "wrong" ones, different choices made at an early stage may result in significantly different design structures and features. Any attempt at producing a "stable" solution will therefore usually involve some iteration in the process of decomposition, in order to explore the effects of different choices.

(i) When to stop subdividing

For any process of subdivision, it is necessary to identify some "atomic" level at which any further decomposition provides no useful return. Designing by stepwise refinement alone lacks any firm guidelines on this issue.

(ii) Problems of replication and recombination

A sequence of refinements is likely to produce some duplication of low-level operations. Where more than one designer is involved, these may prove difficult to recognize, but they do need to be resolved.

(iii) Use as a preliminary step in other design methods

Top-down design is often be used as a preliminary step in other design methods, especially where there is a need to separate concurrent components of a system. It may be of value in distinguishing the major modules of a system, and in dividing tasks among a team of designers.

b. Design by composition, evolution of design methods

The emphasis in a top-down approach tends to be upon *operations*, while the composition approach makes use of *entities* or *objects*, modeling these and the operations performed upon them. The trend in the evolution of systematic design methods is toward a greater balance between objects and operations in formulating the description of the solution. (We can regard an emphasis on operations alone as being equivalent to describing a solution by using only verbs and adverbs, whereas adding to this through the use of objects provides us with the nouns needed to provide a fuller description.)

c. Stylized design

In a few domains of application, it may well be that a strongly stylized model of a "good" solution already exists. An example of such a domain is *compiler writing*, in which a number of relatively standard techniques and structures have been evolved for the production of compilers. Where such a model exists, the adoption of a more general design strategy is usually of little value or purpose.

References:

Papers: Wirth71

Books: Bergland81, Cameron83

4. Design Representations

The use of diagrammatic and textual information to represent different viewpoints of a design is an important tool for the designer. While most representations are normally introduced within the context of a particular design method, designers can, and do, make use of representations in a method-independent manner as a part of the problem-solving process. The role of each form should be identified in terms of its level of abstraction, its use for such purposes as supporting modeling during the earlier stages of design, and for providing "blueprints" in the later stages. Indeed, one view of the design "process" is that it consists of a series of *transformations* between "representations." This, in turn, suggests a framework for categorizing forms of representation. Figure 3 shows a classification, in terms of their roles, of the forms listed here.

a. Data flow diagrams (DFD)

These are "directed graphs" in which the nodes represent processing activities and the arcs specify the transfer of information between these. Good references with examples are [DeMarco79] and [Page-Jones80]. More formal realizations are also in use, and examples of such a form (as used with SSADM) are given in [Downs88].

b. HIPO diagrams

HIPO diagrams (Hierarchy-Process-Input-Output) were developed at IBM as design representations for use in the development of software by top-down techniques and for the documentation of released products. For examples see [Fairley85, Martin84].

c. Structure charts

These are tree-like diagrams used to represent the run-time calling hierarchy of the modules forming a sequential program. They were originally developed for use with Structured Systems Analysis and Structured Design [Page-Jones80, Yourdon-79]. A number of variant forms exist, and a somewhat similar form with a different interpretation is also used in JSP [Cameron83, Ingevaldsson86, Jackson75], where it is termed a Structure Diagram. (See also Entity Structure Diagrams, below.)

Phase	Purpose	Suitable Forms
Architectural Design	Modeling the problem	Data flow diagrams
	Modeling the outline solution	"Block" diagrams Data flow diagrams
Detailed Design	Describing hierarchical structure	Structure chart Entity-structure diagram System structure diagram Pseudocode
	Describing data structure	Structure diagram (JSP)
	Describing the solution logic	Decision table Finite state machine Statechart Pseudocode
	Describing the packaging	Structure graph "Block" diagrams

Figure 3. Classification of representation forms.

d. Decision tables

Used to specify complex decision logic at the level of detailed design. Some examples are given in [Fairley85].

e. Entity structure diagrams (JSD)

Jackson's form of Structure Diagram is also used for modeling the structure of design entities. In particular, it provides information about the time-ordering of the actions performed by an entity. See examples in [Cameron83] and [Sutcliffe88].

f. System specification diagrams (JSD)

This form is used to represent the network of processes in a system, and also the communication links between them. See [Cameron83] and [Sutcliffe88].

g. Entity history diagrams

A form of diagram used in SSADM. They are related to the entity-structure form used in JSD to represent the time-ordered actions of a design entity. See [Downs88].

h. Structure graphs

A form of block diagram used to describe the *packaging* of the elements of a system [Buhr84]. The form is strongly oriented to use with the **Ada** programming language, although it is also useful with programming languages such as **Modula-2**.

i. Finite state machines

Used to specify operations in terms of sets of inputs and outputs, sets of states, and functions. Can be used at all levels of design abstraction. [Birrell85] gives some simple examples.

j. Statecharts

Devised by Harel for use in describing large and complex reactive system [Harel88]. Their claimed advantage over mechanisms such as Finite State Machines includes the provision of hierarchy, a brevity of form, and the ability to describe concurrent operations.

k. Petri nets

These are used to model the interactions of concurrent systems by showing causal relationships. See [Birrell85] for examples.

l. Pseudocode

Used for describing sequential algorithms for detailed program structures. Sometimes termed "Structured English."

m. Formal design languages

Mathematically-based languages may be used to facilitate formal reasoning about the properties of a design. Algebraic forms such as OBJ-2 [Futatsugi85] provide a property-oriented description which is based upon the definition of objects and operations upon them described in terms of axioms. The model-oriented forms such as VDM [Bjørner82, Pedersen88] and Z [Hayes87] use a description based upon defining objects and operations built from a basic set of pre-defined types and their characteristic operations.

References:

Papers: Bergland81, Harel88, Stevens74

Books: Birrell85, Bjørner82, Buhr84, Cameron83, DeMarco79, Downs88, Fairley85, Hayes87, Martin84, Pedersen88, Peters81, Sutcliffe88, Yourdon79

III. Design Practices: Design Methods

The sections in this part of the module provide outline descriptions of a selection of widely-used design methods that represent a range of principles, features and forms.

1. Structured Systems Analysis and Structured Design

The related techniques of Structured Systems Analysis [DeMarco79, Gane79] and Structured Design [Page-Jones80, Yourdon79] together form a system design technique based upon decomposition. However, they make use of *data-flow* considerations, rather than being based solely upon function as a criterion.

a. Problem decomposition (SSA)

The analysis step is an initial model-building process based upon the use of Data Flow Diagrams (DFDs). These are drawn, expanded and analyzed in this initial step. The initial DFDs may describe existing *physical* processes, but the final forms should be concerned only with *logical* processes that occur within the system. The basic steps are listed below:

- Draw data flow diagrams.
- Refine and evaluate DFDs.
- Check DFDs for consistency and for data conservation.

b. Create a data dictionary

The *Data Dictionary* is concerned with recording the information content of data, rather than with its physical realization. It augments the DFD by defining any data forms mentioned in the DFD, including data flows, data used within processes, and any components of these. Describing the data, correlating the data dictionary to the data flow diagrams, the treatment of aliases, the definitions in the data dictionary, data structures, data flows and data stores, and the implementation of a data dictionary are all problems to be dealt with at this stage. Steps to be performed are:

- Describe the data in a *logical* form.
- Describe data structures, data flows, and data stores.
- Correlate the data dictionary with the data flow diagram.
- Data dictionary implementation.

c. Describe process logic

In this phase, the designer is concerned with describing the operations involved in the DFD (that is, the actions implied by the 'bubbles'). Issues involved are:

- Analyzing and presenting process logic.
- Use of Structured English or pseudocode.

d. Deriving a structured design from the logical model

This phase involves deriving the hierarchical program design from the non-hierarchical DFD. The steps are:

- Control considerations.
- Changeability considerations/Domain of change.
- Module coupling/Cohesion, binding.
- Transform Analysis.
- Refining the design, including use of design heuristics.
- Error and exception handling.
- Use of Transaction Analysis.

Teaching Consideration: *Before describing the process, it may be useful to reiterate the quality issues that are involved when making choices between design options. The process of* transform analysis *can then be used in conjunction with these, and the resulting need to refine the design, adding such features as exception handling, should be described. The supplementary technique of* transaction analysis *should be discussed.*

e. An assessment of SSA/SD

(i) Domains of application

This technique is primarily oriented toward the design of sequential programs, although in principle there is potential for basing a concurrent design upon the DFD. This, together with the emphasis upon information flow, has led to this method being widely used in data processing systems.

(ii) Major strengths

The DFD is readily comprehensible to the end user, who can therefore provide direct input to the design process. The method is generally well-documented and is supported by a number of software tools, primarily graphics editors.

(iii) Major weaknesses

The steps involved in transform analysis and transaction analysis appear to draw strongly upon heuristic knowledge and lack clear procedural guidelines for such operations as locating the central transform. More generally, the emphasis is upon data *flow* rather than upon the encapsulation of data structures.

References:

Papers: Stevens74

Books: Connor85, DeMarco79, Gane79, Linger79, Myers78, Page-Jones80, Yourdon79

2. Design by Modeling the Problem: JSP and JSD

a. JSP and JSD principles and the relationship between the two methods

These methods are based upon a process of *composition*. JSP (Jackson Structured Programming) is a program design method concerned with smaller, largely sequential, problems. It is highly prescriptive in its form. JSD (Jackson System Development) extends the philosophy used in JSP into a larger domain of application. Its model is based upon a set of disconnected processes. Time-ordering is an important dimension for both methods.

b. JSP design process

The main steps in JSP are listed below:

- Describe data streams using structure diagrams.
- Merge these to create the program structure diagram.
- List operations and allocate to elements in the program structure.
- Convert the program to text without conditions.
- Add iteration/selection conditions.

Teaching Consideration: *JSP should be introduced by using a fairly simple sequential problem. The syntax and semantics of the Structure Diagrams will need to be reiterated.*

c. Handling multiple inputs in JSP—structure clashes

When there are multiple data streams used for input, it is quite possible that these may be organized and structured using different "keys" (an *ordering* clash) or that one of the streams may contain multiple record types (which can lead to an *interleaving* clash). JSP has techniques to help cope with these and with some of the other forms of *structure clash* that occur.

d. Program inversion

A simple JSP design assumes a program that reads and writes a set of serial data streams. The technique of *program inversion* is used to reorganize the design about one or more of its data streams, so as to allow the program to be suspended and resumed, to provide a "conversational" form of operation.

e. An assessment of JSP

(i) Domains of application

JSP is used for the design of sequential programs. Because of the emphasis upon data structure, it has been used widely for the design of data processing systems, although techniques such as program inversion make it possible to use JSP for a wide range of program forms.

(ii) Major strengths

The method is highly *prescriptive*. Hence, different designers can be expected to produce similar designs.

(iii) Major weaknesses

The method becomes too complex for larger systems that have many structure clashes.

f. Extension of philosophy to JSD

The JSD method makes use of an entity-action-attribute model of the world, and this is built up and connected to the "real" world through a series of well-defined steps and operations.

g. Concept of entity-action-attribute

The core of this method involves modeling the problem in terms of a set of *entities* and their *actions*, and of the *attributes* associated with these actions. In the later steps, the method extends the model to include the interactions between entities, as well as between entities and the world external to the model. It models the timing issues involved in these.

h. Steps involved in JSD

Note that JSD is still evolving. The form described in [Jackson83] has been revised slightly. The form described here is taken from [Cameron83] and [Sutcliffe88]:

- Entity/Action and Entity Structure step.
- Initial model step.
- Interactive function step.
- Information function step.
- System timing step.
- Implementation.

i. An assessment of JSD

(i) Domains of application

Primarily intended for large (and possibly concurrent) systems where the time-ordering of events is important. Both data processing and process control systems are suggested as appropriate domains of application.

(ii) Major strengths

The method is relatively prescriptive for the model-building (analysis) steps. It encourages the use of abstraction and makes good use of an "object-oriented" philosophy. While the later steps are less prescriptive, they still provide a well-defined framework for the processes involved.

(iii) Major weaknesses

The JSD approach breaks down for data structures and relationships that cannot reasonably be described in terms of histories of events. The overhead of the method is too large to make its use with small problems worthwhile, and learning enough about the method to be able to make full use of it takes a long time.

References:

Papers: Cameron86
Books: Cameron83, Connor85, Ingevaldsson86, Jackson83, Sutcliffe88

3. Object-Oriented Design (OOD)

The term "Object-Oriented" has acquired a number of meanings and interpretations, both for programming and for design. There is no one clear definition of exactly what is meant by the term "Object-Oriented Design." This degree of variation in the use of the term should be borne in mind when reading any literature on this topic. For the purposes of this module, Object-Oriented design is presented as a method for modeling a problem by taking a balanced view about *objects* and the operations performed upon them, along the lines suggested by Booch [Booch86, Booch87]. By being *data*-oriented this method differs from the previous methods which are all essentially *process*-oriented, even when using data flow or data structure to aid in identifying the processes and their forms.

a. The concept of *objects*

Software-based systems can be modeled in terms of *objects* and *operations*, rather than in terms of *data* and *procedures*. Although it is oriented toward the Smalltalk-80 system, a useful reference here is [Robson81], which makes this distinction very clearly.

b. Abstraction, information-hiding, modularity and localization

These issues should be drawn out within the context of OOD. A useful collection of references is available as [Peterson87a] and [Peterson87b]. Note particularly the 1984 paper by Shaw [Shaw84].

c. Steps involved in object-oriented design

The basic design steps of OOD are:

- Define an informal strategy for the problem solution.
- Identify the objects used in the informal strategy.
- Identify the operations on the objects used in the informal strategy.
- Define the software system architecture and interfaces to the operations.
- Iterate the above process as needed.

Teaching Consideration: *OOD should be described with the aid of an example. [Booch86] gives some useful ideas about this.*

d. Limitations/problems in applying object-oriented design

This method needs an initial solution modeling stage to help with formulating the informal strategy. In the absence of any specific guidelines for the method, this task is likely to draw upon the analysis techniques of other methods.

e. An assessment of OOD

(i) Domains of application

A fairly general range of applicability, although there seem to be few examples of its use for data processing problems.

(ii) Major strengths

Well-matched to current developments in the form of imperative programming languages such as Ada, Modula-2, and C++.

(iii) Major weaknesses

The form of the solution is overly dependent upon the structure of the initial informal strategy, and the method does not provide enough support for the initial identification of objects.

References:

Papers: Abbott83, Booch86, Rentsch82, Robson81, Shaw84

Books: Booch87, Peterson87a, Peterson87b, Wiener84

4. Some Other Systematic Design Methods

The design methods described in the preceding sections are examples of the application of particular philosophies of design. All are quite widely used. However, there are other design methods in use, and this section provides brief summaries of some of them that are widely-used and well-documented. The list is in no way intended to be exhaustive or definitive. It is important, however, that the student be aware that other systematic methods do exist.

Teaching Consideration: *Some of these design methods can be used to illustrate particular points in class. Due to the background of instructor or students, some of these methods may be more appropriate topics of study than aforementioned design methods.*

a. Structured Analysis and Design Technique (SADT)

SADT takes a data-flow view of what are essentially top-down analysis and design activities, using a somewhat individual notation based on *actigrams* and *datagrams*. In particular, this method contains a strong project organization element. Some brief but good examples are given in [Birrell85] and [Fairley85], and a complete and detailed description is available from [Marca88].

b. The Warnier-Orr approach

Warnier's Logical Construction of Programs (LCP), his Logical Construction of Systems (LCS), and the Warnier-Orr Structured Systems Design technique all have roots and philosophy similar to those of JSP/JSD. The diagrammatic form used in LCP is somewhat different from that of JSP, but like JSP, this method is highly prescriptive in form [Orr77, Warnier80].

c. Stepwise Refinement with Verification (Mills)

This method is described in [Linger79] and places emphasis upon the use of mathematical forms, rather than graphical notations. (It is not a "formal method," in the accepted sense of that term, however.) Both data abstraction and function abstraction are used in the design process, and the use of mathematical notation assists with the verification phase. [Linger79] is primarily concerned with detailed design, while [Mills86] addresses the issues of more general problem decomposition.

d. SSADM

SSADM (Structured Systems Analysis and Design Method) provides an example of a highly prescriptive form of design method, using three viewpoints of data and providing explicit means for cross-checking among them. Developed in the U. K., this method is acquiring a published "standard," rather akin to the former registration accorded "Ada". [Downs88] provides a very clear description of the method, together with some examples of its use.

References:

Papers: Bergland81

Books: Bergland81, Birrell85, Connor85, Downs88, Fairley85, Freeman80, Linger79, Marca88, Mills86, Orr77, Peters81, Riddle79, Warnier80

IV. Review of Design Practices

The purpose of the final part of the module is to draw together, summarize, and analyze the content of the previous parts. From this it is possible to give some guidance on the question of "suitability for purpose" for methods when used in particular domains of application, and also to identify the current trends in design practices.

1. Some Assessments of Design Methods

This section presents a general assessment of the design methods currently in use. The extent to which particular methods encourage the designer to produce designs that are "well-structured," in terms of the generally-accepted design principles, is considered as a useful and important feature to be included in this assessment. Criteria for choosing a design method for a particular problem classification, including specific issues that apply to each class of problems, are considered.

a. General assessment of methods

Considerations for design method assessment:

- Design philosophy and selection criteria.
- Prescriptive elements of the method.
- Suitable domains of applicability.
- Scope for direct assessment of quality in the method.

Teaching Consideration: *Design methods should be discussed in terms of the above. An "evaluation matrix" as used in [Blank83] may prove helpful in presenting this material, but because different methods are suited to different domains of application, it is neither practical nor useful to attempt any form of comparative assessment of methods in the sense of attempting to rank them.*

b. The use of software tools to support design methods

There are a number of tools available for use in developing designs using particular design methods. Most of these tools are relatively "passive," in that they provide facilities to support the "representational" part of a method (such as graphics editors), without providing any form of support for the "process" part.

c. Selecting appropriate design approaches for classes of problems

The matching of design methods with design problems is something of a "horses for courses" situation, in that selecting the most appropriate method (approach) will depend upon many factors, including

- the nature of the problem;
- company/project practices;
- available experience with particular design methods;
- scope for reusing existing design components; and
- other personal/local factors.

In making trade-offs between these factors, a designer needs to have an appreciation of any overhead that particular choices may involve (for example, the amount of time needed to learn, say, JSD, as against Structured Design; the likelihood of being able to reuse that knowledge of a design method; the benefits that might be gained from using available software tools; etc.).

References:

Books: Bergland81, Connor85, Freeman80, Peters81, Riddle79

2. Trends and Developments

The evolution of design methods should be reviewed, examining the changes that have occurred in terms of criteria used, basic cognitive models, design representations, domain of applicability etc. This section should consider the relationship between *systematic* and *formal* design methods, and should examine the trends for both of these approaches.

a. Evolution of design methods

While it may be difficult to place specific dates on design methods, since they usually evolve over a period of time, it may be useful to consider a chart showing the historical development of design methods and the major influences upon each one. As with programming languages, some methods are essentially dead-ends, while others may spawn a number of developments that take an idea and expand upon it.

b. Trends in the development of design methods

Some current trends are toward using a more balanced approach to problem modeling and also toward using a greater degree of mathematical formalism. A relatively new influence is that of cognitive science, which considers the modeling process from the viewpoint of the modeler and his ability to handle particular concepts. Developments in imperative programming languages (in Ada, Modula-2, C++) are having significant influence on design methods as well.

References:

Papers: Yau86

Books: Freeman80

Glossary

abstraction

A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information [IEEE83].

batch

A form of processing in which input data streams are processed sequentially and with no interaction with any form of user. Contrast with **reactive**.

cohesion

The degree to which the tasks performed by a single program module are functionally related. Contrast with **coupling** [IEEE83].

complexity

The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics [IEEE83].

concurrent processes

Processes that may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrence of an external event [IEEE83].

coupling

A measure of the interdependence among modules in a computer program. Contrast with **cohesion** [IEEE83].

data abstraction

The result of extracting and retaining only the essential characteristic properties of data by defining specific data types and their associated functional characteristics, thus separating and hiding the representation details. See also **information hiding** [IEEE83].

data dictionary

A collection of the names of all data items used in a software system, together with relevant properties of those items; for example, length of data item, representation, etc. [IEEE83].

design method

A systematic approach to creating a design, consisting of the ordered application of a specific collection of tools, techniques, and guidelines [IEEE83].

exception

An event that causes suspension of normal process execution [IEEE83].

generic software components

System elements that are parameterized in such a way that they can be used with different data objects without its being necessary to modify their form.

HIPO diagrams

Hierarchy-Process-Input-Output diagrams, developed at IBM to help represent schemes for top-down software development, and as aids to the documentation of released products.

information hiding

The technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the inner workings of the module; thus, each module is a "black box" to the other modules in the system [IEEE83].

JSD

Jackson System Development [Cameron83, Sutcliffe88].

JSP

Jackson Structured Programming [Cameron83]

metric

A parameter used as a measure of some program or system attribute, usually concerned with assessing *quality*.

modularity

The extent to which software is composed of discrete components such that a change to one component has minimal impact on other components [IEEE83].

product

An entity designated for delivery to a user.

prototype

A component of a software development cycle that is used for evaluation purposes. For a fuller discussion, see [Budde84].

reactive

A form of processing in which the program's operations are determined through interaction with external processes. Contrast with **batch**.

real-time

Pertaining to the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process. This term is also used to describe systems operating in conversational mode, and processes that can be influenced by human intervention while they are in progress [IEEE83].

reusability

The extent to which a module can be used in multiple applications [IEEE83].

SADT

Structured Analysis and Design Technique. The two main forms of diagram used with this are the *actigram* and the *datagram*. For examples, see [Fairley85, Marca88].

software life cycle

A typical sequence of phased activities that represent the various stages of engineering through which a software system will normally pass.

software maintenance

The process of modifying a product after delivery in order to correct faults (corrective maintenance), to improve performance or other attributes (perfective maintenance), or to adapt the product to a changed environment (adaptive maintenance).

SSADM

Structured Systems Analysis and Design Method, derived from work done by Learmonth, Burchett Management Systems (LBMS). See [Downs88].

Teaching Considerations

Suggested Schedules

The material in this module can be taught in different ways, depending on the time available. Table 1 at the end of this section presents suggestions for three different schedules. The first requires about 30 lecture hours, as might be available in a full semester course on software design. The second requires about 19 hours and might be appropriate for a course that combines software specification and design. The third requires about 9 hours, as in a course that surveys many aspects of software engineering.

Worked Examples

The demonstration of various design methods through worked examples is valuable in teaching this material. Three example problems frequently referenced in the literature are the following:

1. **Text Formatter.** This is a program that reads in one or more files that contain text, together with embedded formatting commands, and generates a nicely formatted output document. It is an example of a straightforward *batch* process.
2. **Library Record Maintenance.** This is a program that maintains a set of records on book holdings and issues for a library. In this form, it is both *sequential* and *reactive*, but it could be extended to have a concurrent structure.
3. **Elevator Control.** This is an example of a *concurrent* system that contains *reactive* components. It can also be considered as an example of an *embedded* system.

The value of these examples is enhanced if the students see the same problems as examples of requirements specification, design, implementation, and testing. However, because design methods are optimized for different domains of application, the instructor is advised against seeking to use the same example problem for demonstrating different methods. Where only limited time is available for teaching this material, JSP may provide the most convenient basis for a worked example that is concise and illustrates clearly the use of both the "process" and "representational" components of a design method.

Exercises

The most obvious teaching strategy is to single out one design method for extended discussion and exercises. It should be emphasized, however, that students who have taken a single course based upon this material should not be expected to undertake design tasks requiring the detailed use of any design method.

Table 1. Suggested Schedules

Topic	*Full Syllabus* Sections	Hrs	*Medium Syllabus* Sections	Hrs	*Short Syllabus* Sections	Hrs
I. The Role of Software Design						
1. The Design Process	a.-i.	2.0	a.-i.	1.5	a.-i.	1.0
2. Design as a Life-Cycle Phase	a.-d.	1.0	a.-d.	1.0	a.-c.	0.5
3. Principles of Design	a.-b.	2.0	a.-b.	1.5	a.-d.	1.0
	c.-d.	1.0	c.-f.	1.5	e.-h.	1.0
	e.-f.	1.0	g.-h.	1.0		
	g.-h.	1.0				
Subtotal for Part I		8.0		6.5		3.5
II. Design Practices: General Issues						
1. Role of Design Methods	a.-d.	1.0	a.-d.	0.5	a.-d.	0.5
2. Problem Classification	a.-d.	1.0	a.-d.	0.5	a.-d.	0.5
3. Design Strategies	a.-c.	1.0	a.-c.	0.5	a.-c.	0.5
4. Design Representations	a.-m.	3.0	a.-m.	2.0	a.,c.	0.5
					e.,h.,l.	0.5
Subtotal for Part II		6.0		3.5		2.5
III Design Practices: Design Methods						
1. Structured Analysis and Design	a.-e.	2.0	a.-e.	1.0	a.-e.	1.0
1a. Worked Examples of Structured Design	*	*	*	*		0.0
2. Jackson's Methods	a.-i.	2.0	a.-i.	1.5	a.-f.	1.0
2a. Worked Examples using Jackson's JSP Method	*	*	*	*		0.0
3. Object-oriented design	a.-e.	1.5	a.-e.	1.0	a.-e.	0.5
3a. Worked Examples of Object-Oriented Design	*	*	*	*		0.0
4. Other Systematic Design Methods	a.-d.	1.0	a.-d.	1.0	a.-d.	0.0
4a. Examples of other methods	*	*	*	*		0.0
Subtotal for Part III		6.5		4.5		2.5
IV Review of Design Practices						
1. Some Comparisons	a.-c.	1.0	a.-c.	0.5	a.-c.	0.5
2. Trends and Developments	a.-b.	0.5	a.-b.	0.5	a.-b.	0.0
Subtotal for Part IV		1.5		1.0		0.5
* Total time devoted to worked examples		4.0-8.0		3.5		0.0
TOTAL TIME		26.0-30.0		19.0		9.0

Comments:

This table is intended to serve as a guideline for the relative amount of time to be spent on each topic.

Full Syllabus	*The instructor could spend a total of 4 to 8 hours on some, but not necessarily all, the worked examples.*
Medium Syllabus	*Most of the topics are covered, but less time is spent on each topic than in the full syllabus. Only one of the worked examples is discussed.*
Short Syllabus	*A number of topics are omitted, and the worked examples are not discussed.*

Bibliographies

Textbooks

This section describes textbooks and reports covering various aspects of software engineering. All of them contain material relevant to software design, and a number of the entries are primarily devoted to describing specific methods used in software design. The annotations provided are mostly in two parts—the first part providing general comments on content, the second part containing suggestions of how the reference may be used for teaching.

Abbott86

Abbott, R. J. *An Integrated Approach to Software Development.* New York: John Wiley, 1986. ISBN 0-471-82646-4.

A general text on software engineering that is organized as a collection of annotated outlines for technical documents that are important to the development and maintenance of software.

Allworth87

Allworth, S. T., and R. N. Zobel. *Introduction to Real-Time Software Design, 2nd Ed.* New York: Springer-Verlag, 1987. ISBN 0-387-91307-6.

One of the few books that is devoted to this particular and rather specialized aspect of software design. The book makes good use of the concept of a *virtual machine* for design of such systems and is well provided with diagrams. Much of the discussion is concerned with detailed design issues. The book pulls together into a single theme material taken from diverse areas.

The instructor might find this a useful text to refer to when looking at domain-specific issues.

Bergland81

Bergland, G. D., and R. D. Gordon. *Software Design Strategies, 2nd Ed.* Washington, D. C.: IEEE Computer Society Press, 1981. ISBN 0-8186-0389-5.

Several major design strategies are developed and compared in this tutorial text, including functional decomposition, Jackson's JSP method, and data flow design. The process of organizing and coordinating the efforts of the design team and descriptions and use of several design tools currently used in industry are also presented. Contains a useful key word index at the end of the book.

Essential collection of papers for the instructor. Contains some useful reading material for students.

Berztiss87

Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Berztiss88

Berztiss, A., and M. A. Ardis. *Formal Verification of Programs.* Curriculum Module SEI-CM-20-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Birrell85

Birrell, N. D., and M. A. Ould. *A Practical Handbook for Software Development.* New York: Cambridge University Press, 1985. ISBN 0-521-25462-0.

Provides a good overview of the software engineering view of system development, supported by an overview of a wide range of the techniques that are available to support each phase of development. The latter half of the book covers a wide range of design issues, together with examples. The book makes particularly good use of diagrams to help make its points.

A good book for students to use in an introductory course on design.

Bjørner82

Bjørner, D., and C. B. Jones. *Formal Specifications and Software Development.* Englewood Cliffs, N. J.: Prentice-Hall, 1982.

The primary concern of this text is the development of formal specifications, with emphasis being placed upon the need to be able to relate design to specification. It contains chapters by a number of authors describing aspects and applications of VDM (*Vienna Development Method*), presented at an advanced level and requiring some background in discrete mathematics.

A book for the instructor rather than for the student.

Blank83

Blank, J. and M. J. Krijger, eds. *Software Engineering: Methods and Techniques.* New York: Wiley-Interscience, 1983. ISBN 0-471-88503-7.

A report produced by the Information Structures Subgroup of the Dutch Database Club, which aims to evaluate and compare a number of different de-

sign methods. Many of the methods will be unfamiliar to most readers, although the list does include more widely-known methods such as SADT, Warnier-Orr, and JSD. A summary of the features of each method is included.

The use of an "Evaluation Matrix" as a means of presenting information about the features and application areas of a method is an interesting feature. It can provide useful material for the instructor.

Boehm81

Boehm, B. *Software Engineering Economics.* Englewood Cliffs, N. J.: Prentice-Hall, 1981. ISBN 0-13-822122-7.

Boehm is particularly well known for his COCOMO cost estimation model. This book provides a practical introduction to the planning and estimation tasks that are involved in software development. It provides a somewhat different insight into the design process and trade-offs and related issues that are associated with it. The book is centered around a description of COCOMO, which provides the framework for discussing a large number of issues that are important for both the design process and the design product.

An essential reference text for both instructor and student.

Booch87

Booch, G. R. *Software Engineering with Ada, 2nd Ed.* Menlo Park, Calif.: Benjamin/Cummings, 1987. ISBN 0-8053-0600-5.

Describes the Ada language and its use, with particular reference to the features of Ada that support software engineering principles. Contains five examples on object-oriented design, presented in a highly readable form.

The examples of object-oriented design provide some valuable ideas and source material for the instructor.

Brooks75

Brooks, Jr., F. P. *The Mythical Man-Month.* Reading, Mass.: Addison-Wesley, 1975. ISBN 0-201-00650-2.

This book can be regarded as being a classical presentation of the problems that may be encountered in the development and management of a large software system. As such, it should be regarded as essential preliminary reading for anyone who has little or no prior experience of *programming-in-the-large*, or who has not been involved in project management. The book contains many important lessons for the designer, presented in a particularly readable format.

This book should be part of the prerequisite reading for every student who aspires to study software engineering.

Budde84

Budde, R., K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, eds. *Approaches to Prototyping.* New York: Springer-Verlag, 1984. ISBN 0-387-13490-5.

A collection of papers from a workshop held to study the use of different forms of prototyping in systems design and development. The opening paper gives a good review and taxonomy for the field.

Can provide useful reference material for both instructor and students.

Buhr84

Buhr, R. J. A. *System Design with Ada.* Englewood Cliffs, N. J.: Prentice-Hall, 1984. ISBN 0-13-881623-9.

Presents and illustrates a top-down, design-oriented introduction to Ada, using a specially developed graphical design notation (the *structure graph*). Presentation is oriented toward concurrent programs.

Recommended reading for the instructor. Good material for student reading and student class presentations.

Cameron83

Cameron, J. R. *JSP & JSD: The Jackson Approach to Software Development.* Washington, D. C.: IEEE Computer Society Press, 1983. ISBN 0-8186-8516-6.

A collection of articles and papers describing JSP and JSD and illustrating these methods using a range of examples of reasonable size and complexity.

Good source material for the instructor. A potential source of material for student tutorials.

Connor85

Connor, D. *Information System Specification and Design Road Map.* Englewood Cliffs, N. J.: Prentice-Hall, 1985. ISBN 0-13-464868-4.

Essentially aimed at DP-style systems that are concerned with record management. Gives an overview of a number of methods based on a document library problem.

Cross84

Cross, N., ed. *Developments in Design Methodology.* New York: John Wiley, 1984. ISBN 0-471-10248-2.

A comprehensive summary of work in the field of design theory over the past twenty-five years. Includes important papers by J Christopher Jones, Christopher Alexander, Herbert Simon, and Horst Rittel.

Davis83

Davis, W. S. *Systems Analysis and Design.* Reading, Mass.: Addison-Wesley, 1983. ISBN 0-201-10271-4.

A presentation on analysis and design based around the use of three case studies. Each of the case studies is taken through the steps of problem definition, feasibility study, analysis, system design, and detailed design. The main emphasis of the book is on analysis rather than design, as such. The book is oriented toward business applications. The book primarily makes use of the SSA/SD approach to design.

The case studies may provide a useful basis for class discussions.

DeMarco79

DeMarco, T. *Structured Analysis and System Specification.* Englewood Cliffs, N. J.: Yourdon Press, 1979. ISBN 0-917072-07-3.

A readable book on structured analysis and system specification that covers data flow diagrams, data dictionaries, and process specification.

Good source material for the instructor. Recommended reading for students.

Downs88

Downs E., P. Clare, and I. Coe. *SSADM: Structured Systems Analysis and Design Method.* New York: Prentice-Hall, 1988. ISBN 0-13-854324-0.

SSADM is a highly prescriptive design method, with a fully defined structure and terminology. This book begins by describing the structure of the method, and then describes the activities that should be associated with each of the *phases*, *stages*, *steps*, and *tasks* involved. Written in a clear and readable style, this book makes good use of diagrams throughout.

The level of detail makes this book more suitable for use by the instructor than by the student, unless SSADM is being used as the main topic of a course unit.

Fairley85

Fairley, R. E. *Software Engineering Concepts.* New York: McGraw-Hill, 1985. ISBN 0-07-019902-7.

Describes the basic concepts and major issues of software engineering, including current tools and techniques. Contains a chapter on design that covers fundamental design concepts, including assessment criteria, design notations, and design techniques.

Good source material for instructor. Recommended reading for students.

Fox82

Fox, J. M. *Software and Its Development.* Englewood Cliffs, N. J.: Prentice-Hall, 1982. ISBN 0-13-822098-0.

Discusses the development of large scale software.

Franta82

Franta, W. R., H. K. Berg, W. E. Boebert, and T. G. Moher. *Formal Methods of Program Verification and Specification.* Englewood Cliffs, N. J.: Prentice-Hall, 1982.

Freeman80

Freeman, P., and A. I. Wasserman, eds. *Software Design Techniques, 4th Ed.* Silver Spring, Md.: IEEE Computer Society Press, 1980. ISBN 0-8186-0514-0.

A large collection of papers covering basic concepts, analysis and specification, architectural design, data design, detailed design, and management issues. Includes several of the papers listed in this bibliography section.

Should provide a useful source of material for the instructor and useful material for student tutorials.

Gane79

Gane, C., and T. Sarson. *Structured Systems Analysis: Tools and Techniques.* Englewood Cliffs, N. J.: Prentice-Hall, 1979. ISBN 0-13-854547-2.

One of the more widely used books on structured systems analysis. The book discusses some of the problems in analysis, reviews graphical tools, and shows how the graphical tools fit together to make a logical model. Each tool is treated in detail, including the data flow diagram. A structured system development method that takes advantage of the tools is presented. The importance of changeability and how it may be treated is also covered.

Essential instructor reading. Recommended student reading.

Hansen86

Hansen, K. *Data Structured Program Design.* Englewood Cliffs, N.J.: Prentice-Hall, 1986. ISBN 0-9605884-2-6.

The main theme of this book is Orr's *Data Structured Systems Development* (DSSD) method, which is also compared and contrasted with the related work of Warnier and Michael Jackson (JSP). The program examples use COBOL, although a knowledge of this language is probably not essential to an understanding of the material. The book contains many examples of the use of Warnier/Orr diagrams.

Written in a very readable style. It may be rather detailed in its treatment of the subject matter for use by students, but it contains some useful guidelines and ideas for the instructor.

Hayes87

Hayes, I, ed. *Specification Case Studies.* Englewood Cliffs, N. J.: Prentice-Hall, 1987. ISBN 0-13-826579-8.

A collected set of case studies that are all based upon the use of Z, providing a well-structured introduction to the use of formal methods. The section on specification of the UNIX filing system may involve sufficiently familiar material to provide a good introduction for many students.

Suitable for use by both instructors and students.

IEEE83

IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* New York: IEEE, 1983. ANSI/IEEE Std 729-1983.

Provides definitions for many of the terms used in software engineering.

Ingevaldsson86

Ingevaldsson, L. *JSP: A Practical Method of Program Design, 2nd Ed.* Bromley, Kent, U. K.: Chartwell-Bratt Ltd., 1986. ISBN 0-86238-107-X.

A practical book that relates JSP concepts to a wider domain. (The reader is invited to draw structure diagrams to describe a train, a telephone directory, and other structures). This book is in a very readable style, and is well-provided with examples and exercises (and with solutions for the latter).

A useful book for anyone teaching any details about JSP, and well-suited for use by students.

Jackson75

Jackson, M. A. *Principles of Program Design.* Orlando, Fla.: Academic Press, 1975. ISBN 0-12-379050-6.

Presents a semiformal approach to program design that maps the syntactic structure of a program's input into a structure for an algorithm to process that input. This can be considered as the sourcebook for JSP, and despite the use of COBOL for the programming examples, it discusses a lot of important issues.

For a more general approach to the use of JSP, see [Cameron83]. [Ingevaldsson86] is better suited for student use.

Jackson83

Jackson, M. A. *System Development.* Englewood Cliffs, N. J.: Prentice-Hall, 1983. ISBN 0-13-880328-5.

This book contains the original description of JSD. It is built around three worked examples. Note that [Cameron83] and [Sutcliffe88] provide descriptions of a more current form of the JSD method and contain more manageable examples for students.

A source of material for the instructor, rather than for the student.

Jensen79

Jensen, R. W., and C. C. Tonies, eds. *Software Engineering.* Englewood Cliffs, N. J.: Prentice-Hall, 1979. ISBN 0-13-822130-8.

A collection of articles that are primarily oriented toward management. However, structured program design is covered.

Jones70

Jones, J. Christopher. *Design Methods: Seeds of Human Futures.* New York: Wiley Interscience, 1970. ISBN 0-471-44790-0.

This is a quite widely-cited book. It treats design as a strategy for *problem-solving* in a fairly wide domain, rather than being centered on the design of software. It is included here because it is an example of a book that emphasizes the interdisciplinary nature of design, and so illustrates the point that the problems we encounter are not unique to software design. It also highlights cognitive issues of the design process.

A book that offers thoughts and ideas for the instructor, and which might also provide some thoughts for the student, when used for background reading.

Jones80

Jones, C. B. *Software Development: A Rigorous Approach.* Englewood Cliffs, N. J.: Prentice-Hall, 1980. ISBN 0-13-821884-6.

Presents a formal approach to specification and verification of programs and to the use of abstract data types.

The material of this book may be difficult for anyone who lacks the necessary mathematical background or who is unfamiliar with the type of notation used.

Kernighan76
Kernighan, B. W., and P. Plauger. *Software Tools.* Reading, Mass.: Addison-Wesley, 1976. ISBN 0-201-03668-1.

A popular guide to programming style and to the organization and design of software tools. Strongly linked to the UNIX philosophy of providing small, independent tools and linking these together to produce more powerful tools tailored for specific purposes.

Provides a readable and interesting source of ideas for the student, taking a somewhat different view of design than that used in most of the texts listed in this bibliography.

Linger79
Linger, R. C., H. D. Mills, and B. I. Witt. *Structured Programming: Theory and Practice.* Reading, Mass.: Addison-Wesley, 1979. ISBN 0-201-14461-1.

Central theme is the design of mathematically correct structured programs by the use of systematic methods of program analysis and synthesis.

Instructors may find this book useful for material on structured programming. Material may be appropriate for student tutorials.

Liskov86
Liskov B., and J. Guttag. *Abstraction and Specification in Program Design.* New York: McGraw-Hill, 1986. ISBN 0-07-037996-3.

Discusses different uses of abstractions, based largely around the programming language CLU, and with an emphasis upon the issues of programming-in-the-large. Primarily concerned with relatively detailed design issues.

Marca88
Marca, D. A., and C. L. McGowan. *SADT: Structured Analysis and Design Technique.* New York: McGraw-Hill, 1988. ISBN 0-07-040235-3.

A detailed description of SADT, which makes use of a generous supply of illustrations and examples, as well as providing a number of case studies taken from different application domains. The large size format used for the book makes the examples particularly clear and readable.

The level of detail provided makes this particularly suitable for use as a source of material for the instructor.

Martin84
Martin J., and C. McClure. *Diagramming Techniques for Analysts and Programmers.* Englewood Cliffs, N. J.: Prentice-Hall, 1984. ISBN 0-13-208794-4.

A useful summary of some major forms of diagrams that also provides a set of examples for a wide range of diagrammatic forms.

Useful material for the instructor.

Millington81
Millington, D. *Systems Analysis and Design for Computer Applications.* New York: Halsted Press, 1981. ISBN 0-470-27224-4.

Mills86
Mills, H. D., R. C. Linger, and A. R. Hevner. *Principles of Information Systems Analysis and Design.* Orlando, Fla.: Academic Press, 1986. ISBN 0-12-497545-3.

This book presents a *box structure* approach to the design of information systems, based upon the use of "black box," "state machine," and "clear box" structures. Management issues involved in the design process are included in the presentation, although the main emphasis is on the design transformation techniques involved.

Mills88
Mills, E. E. *Software Metrics.* Curriculum Module SEI-CM-12-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Myers78
Myers, G. J. *Composite Structure Design.* New York: Van Nostrand, 1978. ISBN 0-442-80584-5.

A data flow approach to program design similar to Yourdon79.

Page-Jones80
Page-Jones, M. *The Practical Guide to Structured Systems Design.* Englewood Cliffs, N. J.: Yourdon Press, 1980. ISBN 0-917072-17-0.

Presents the tools of structured analysis and shows how to use these tools. Defines the activity of de-

sign and the qualities of a good design with respect to partitioning, coupling, and cohesion. Presents a discussion on transform and transaction analysis.

A readable book that should be a valuable source of material for both the instructor and student interested in a comprehensive presentation of structured systems analysis.

Pedersen88
Pedersen, J.S. *Software Development Using VDM.* Curriculum Module SEI-CM-16-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., April 1988.

Peters81
Peters, L. J. *Software Design: Methods and Techniques.* Englewood Cliffs, N. J.: Yourdon Press, 1981. ISBN 0-917072-19-7.

The first two chapters of this book give a very good description of the software design process, viewed as a problem-solving process. The issues of design representation are also discussed in some detail. The later chapters on design methods are now a little dated, in terms of the selection of methods used.

This book contains a lot of useful material for the instructor, and the student can benefit from using the book as secondary support material.

Peterson87a
Peterson, G. E, ed. *Object-Oriented Computing, Volume 1: Concepts.* Washington, D. C.: IEEE Computer Society Press, 1987. ISBN 0-8186-0821-8.

A useful collection of papers concerned with the development of object-oriented thinking. It also manages to strike a balance between the view of Smalltalk-80 and that of languages such as Ada.

Peterson87b
Peterson, G. E, ed. *Object-Oriented Computing, Volume 2: Implementations.* Washington, D. C.: IEEE Computer Society Press, 1987. ISBN 0-8186-0822-6.

Complements the material of Volume 1 by assembling papers concerned with making use of object-oriented thinking in various forms of systems.

Pressman82
Pressman, R. S. *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill, 1982. ISBN 0-07-050781-3.

A survey that covers the software life cycle in a relatively informal manner. Includes separate chapters on stepwise refinement, cohesion and coupling, data flow, and data structure.

A good source of material for the instructor, could also be used as secondary reading material for students.

Rombach87
Rombach, H. D. *Software Specification: A Framework.* Curriculum Module SEI-CM-11-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Sommerville85
Sommerville, I. *Software Engineering, 2nd Ed.* Reading, Mass.: Addison-Wesley, 1985. ISBN 0-201-14229-5.

General textbook on software engineering, covering the software life cycle and human aspects of software engineering. Some emphasis on Ada.

Good reading and source material for instructor. Suitable reading for students, although the ratio of text to diagrams makes it rather heavy going in parts.

Sutcliffe88
Sutcliffe, A. *Jackson System Development.* New York: Prentice-Hall, 1988. ISBN 0-13-508128-9.

A clear introduction to the concepts and use of JSD. A particularly useful feature is the inclusion of two worked examples at the back of the book.

Warnier80
Warnier, J. D. *Logical Construction of Programs.* New York: Van Nostrand, 1980. ISBN 0-442-22556-3.

Presents a semiformal approach to program design that maps the structure of a program's input into a structure for an algorithm to process the input.

Wiener84
Wiener, R. S., and R. F. Sincovec. *Software Engineering with Modula-2 and Ada.* New York: John Wiley, 1984. ISBN 0-471-89014-6.

Examines each phase of the software engineering process. The focus is on object-oriented design, with implementation in Modula-2 or Ada. Presents a review of design methods and principles.

May be useful for use by an instructor or by a student interested in object-oriented design and implementations in Modula-2 and Ada.

Yourdon79

Yourdon, E., and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design.* Englewood Cliffs, N. J.: Prentice-Hall, 1979. ISBN 0-13-854471-9.

Presents a data flow approach to program design similar to [Myers79]. Much of this material is an expansion of the ideas expressed in [Stevens74]. Well-written and a good source of material for the instructor, although perhaps a little dated for use by the student.

Yourdon85

Yourdon, E. *Structured Walkthroughs, 3rd Ed.* New York: Yourdon Press, 1985. ISBN 0-917072-55-3.

A very readable book that discusses a particular way of managing the *process* of design and assessing the *product.* Reviews can be used with all methods, and this book offers some practical advice about how to organize them.

Students should be encouraged to read this very practical little book to help provide them with a part of the general background to the design process.

Papers

This section provides an annotated set of references to a select group of published papers covering various aspects of software design and design methods. The annotations are structured as for textbooks. Abstracts are included whenever available.

Abbott83

Abbott, R. J. "Program design by informal English descriptions." *Comm. ACM 26,* 11 (Nov. 1983), 882-894.

Abstract: A technique is presented for developing programs from informal but precise English descriptions. The technique shows how to derive data types from common nouns, variables from direct references, operators from verbs and attributes, and control structures from their English equivalents. The primary contribution is the proposed relationships between common nouns and data types; the others follow directly. Ada is used as the target programming language because it has useful program design constructs.

This paper describes the relationship between objects and data types. It introduces the ideas of object-oriented design as a means of deriving the structure of a program, based upon the use of an informal but precise English description. The examples used to illustrate the points are based upon the Ada programming language. They emphasize the way that this approach designs around the concept of modularity.

A useful source of material and examples of object-oriented methods for the teacher, and one that can usefully be read by any student who wants a better and fuller understanding of this approach.

Adelson85

Adelson, B., and E. Soloway. "The Role of Domain Experience in Software Design." *IEEE Trans. Software Eng. SE-11,* 11 (Nov. 1985), 1351-1360.

Abstract: A designer's expertise rests on the knowledge and skills which develop with experience in a domain. As a result, when a designer is designing an object in an unfamiliar domain he will not have the same knowledge and skills available to him as when he is designing an object in a familiar domain. In this paper we look at the software designer's underlying constellation of knowledge and skills, and at the way in which this constellation is dependent upon experience in a domain. What skills drop out, what skills, or interactions of skills come forward as experience with the domain changes? To answer the above question, we studied expert designers in experimentally created design contexts with which they were differentially familiar. In this paper we describe the knowledge and skills we found were central to each of the above contexts and discuss the functional utility of each. In addition to discussing the knowledge and skills we observed in expert designers, we will also compare novice and expert behavior.

One of the very few papers to consider the effects of a designer's prior experience upon the decisions made in particular circumstances. The paper covers such issues as the building of mental models, the representation of constraints, and the use of such techniques as making notes. An important first step in an area that is largely uncharted.

A good source of discussion material and ideas. The paper is sufficiently short and well-presented for it to be read by advanced students. While students may find difficulty in relating to all of the issues, there is much that they should be able to relate to their own experiences.

Belady76

Belady, L. A., and M. M. Lehman. "A model of large program development." *IBM Systems J. 15,* 3 (1976), 225-252.

Abstract: Discussed are observations made on the development of OS/360 and its subsequent enhancements and releases. Some modelling approaches to organizing these observations are also presented.

This very comprehensive paper can be regarded as one of the classic papers in this subject area, in terms of its contribution to our understanding of how the structures of very large systems evolve with time. While it is not strictly concerned with specific design methods or directly with the design process, it does provide an important part of the background material needed for an understanding of the problems that face the designer.

A good source of material for the teacher, and although it is rather long for the purpose, it can also provide a good source of discussion material for student tutorials.

Bergland81

Bergland, G. D. "A Guided Tour of Program Design Methodologies." *Computer 14*, 10 (Oct. 1981), 13-37.

A useful survey of design methods with a slant toward data processing issues. It provides a comparative study of a number of different design methods, based on an example of a stock control problem. It makes its points in a clear and readable manner.

Good source material for the teacher and a good review article for use by students, too, although the restricted domain of the example problem requires that it be supplemented in some way by lectures and further reading.

Booch86

Booch, G. "Object-Oriented Development." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 211-221.

Abstract: Object-oriented development is a partial-lifecycle software development method in which the decomposition of a system is based upon the concept of an object. This method is fundamentally different from traditional functional approaches to design and serves to help manage the complexity of massive software-intensive systems. The paper examines the process of object-oriented development as well as the influences upon this approach from advances in abstraction mechanisms, programming languages, and hardware. The concept of an object is central to object-oriented development and so that properties of an object are discussed in detail. The paper concludes with an examination of the mapping of object-oriented techniques to Ada using a design case study.

A well-presented summary of object-oriented methods, based around the two examples of a car cruise-control system and a navigational/weather data collection buoy. This paper also comments on the relationship between this method and the JSD method.

A useful source of material for the teacher and a paper that can reasonably be read by students who are seeking some insight into this approach to designing systems.

Cameron86

Cameron, J. R. "An Overview of JSD." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 222-240.

Abstract: The Jackson System Development (JSD) method addresses most of the software lifecycle. JSD specifications consist mainly of a distributed network of processes that communicate by message-passing and by read-only inspection of each other's data. A JSD specification is therefore directly executable, at least in principle. Specifications are developed middle-out from an initial set of "model" processes. The model processes define a set of events, which limit the scope of the system, define its semantics, and form the basis for defining data and outputs. Implementation often involves reconfiguring or transforming the network to run on a smaller number of real or virtual processors. The main phases of JSD are introduced and illustrated by a small example system. The rationale for the approach is also discussed.

A clear summary of a rather complicated and powerful design method. As the method is still evolving, the steps described are slightly different from those presented in Michael Jackson's book *System Development.*

Rather too long and complicated for direct use by students, but a good source of material for the teacher.

Futatsugi85

Futatsugi, K., *et al.* "Principles of OBJ-2." *Conf. Record 12th Ann. ACM Symp. on Principles of Programming Lang.* New York: ACM, 1985, 52-66.

OBJ-2 is a functional programming language with an underlying formal semantics that is based upon equational logic and an operational semantics that is based on rewrite rules. The paper deals with issues of modularization and parameterization, as well as implementation techniques.

Harel88

Harel, D. "On Visual Formalisms." *Comm. ACM 31*, 5 (May 1988), 514-530.

Abstract: The higraph, a general kind of diagramming object, forms a visual formalism of topological nature. Higraphs are suited for a wide array of applications to databases, knowledge representation, and, most notably, the behaviour. l specification of complex concurrent systems using the higraph-based language of statecharts.

An elegant and clearly-written paper discussing a

number of important issues about design representation. The first part of the paper is concerned with general issues, whereas the latter part provides an interesting exposition of *statecharts*. The paper includes a detailed example, in the form of a description of a digital watch.

Henderson86

Henderson, P. "Functional Programming, Formal Specification, and Rapid Prototyping." *IEEE Trans. Software Eng. SE-12*, 2 (Feb. 1986), 241-250.

Abstract: Functional programming has enormous potential for reducing the high cost of software development. Because of the simple mathematical basis of functional programming it is easier to design correct programs in a purely functional style than in a traditional imperative style. We argue here that functional programs combine the clarity required for the formal specification of software designs with the ability to validate the design by execution. As such they are ideal for rapidly prototyping a design as it is developed. We give an example which is larger than those traditionally used to explain functional programming. We use this example to illustrate a method of software design which efficiently and reliably turns an informal description of requirements into an executable formal specification.

This paper illustrates a rather different approach to design of software systems, based on prototyping and using formal specifications.

Parnas72

Parnas, D. L. "On the Criteria to be used in decomposing systems into modules." *Comm. ACM 15*, 12 (Dec. 1972), 1053-1058.

Abstract: This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

A truly "classical" paper, in the sense of being often cited but probably rarely read. It is a very important paper that lays down the basic ideas about information hiding but in a very concise and compact form. The discussion is based upon an example of a problem that may not be very familiar to many readers.

The teacher must read this paper; the student might do better to settle for the teacher's interpretation.

Parnas79

Parnas, D. L. "Designing Software for Ease of Extension and Contraction." *IEEE Trans. Software Eng. SE-5*, 2 (March 1979), 128-137.

Abstract: Designing software to be extensible and easily contracted is discussed as a special case of design for change. A number of ways that extension and contraction problems manifest themselves in current software are explained. Four steps in the design of software that is more flexible are then discussed. The most critical step is the design of a software structure called the "uses" relation. Some criteria for design decisions are given and illustrated using a small example. It is shown that the identification of minimal subsets and minimal extensions can lead to software that can be tailored to the needs of a broad variety of users.

An extension to his 1972 paper, in terms of relating the basic ideas developed there to the problems encountered by a programmer. Largely concerned with the relationships between modules, particularly with the concept of the "uses" relationship.

A fairly hard paper for the student, but one that can provide a good source of discussion material.

Rentsch82

Rentsch, T. "Object Oriented Programming." *ACM SIGPLAN Notices 17*, 9 (Sept. 1982), 51-57.

Relates object-oriented design practices to a number of existing systems, including *Smalltalk*. The author emphasizes the multiple views of object-orientation applied to computer systems.

Not suitable for direct use by students, due to its having too many references to a wide range of architectures and programming languages. The teacher might enjoy it, however.

Robson81

Robson, D. "Object-Oriented Software Systems." *Byte 6*, 8 (Aug. 1981), 74-86.

Abstract: This article describes a general class of tools for manipulating information called object-oriented software systems. It defines a series of terms, including software system and object-oriented. The description is greatly influenced by a series of object-oriented programming environments developed in the last ten years by the Learning Research Group of Xerox's Palo Alto Research

Center, the latest being the Smalltalk-80 system. The article describes object-oriented software systems in general, instead of the Smalltalk-80 system in particular, in order to focus attention on the fundamental property that sets the Smalltalk-80 system apart from most other programming environments. The words "object-oriented" mean different things to different people. Although the definition given in this article may exclude systems that should rightfully be called object-oriented, it is a useful abstraction of the idea behind many software systems.

A brief and clear exposition of the distinctive feature of the object-oriented viewpoint, which is contrasted with the more "traditional" viewpoint of data and procedures. Some important concepts that it briefly introduces include **classes** and **instances**, and the concept of **inheritance**.

Shaw84

Shaw, M. "Abstraction Techniques in Modern Programming Languages." *IEEE Software 1*, 4 (Oct. 1984), 10-26.

In this paper, the author looks at programming language responses to the dual problems of high software cost and low software quality. She argues: "The best new developments in programming languages support and exploit abstraction techniques. These techniques emphasize engineering concerns, including design, specification, correctness, and reliability."

Stevens74

Stevens, W. P., G. J. Myers, and L. L. Constantine. "Structured Design." *IBM Systems J. 13*, 2 (May 1974), 115-139.

Abstract: Considerations and techniques are proposed that reduce the complexity of programs by dividing them into functional modules. This can make it possible to create complex systems from simple, independent, reusable modules. Debugging and modifying programs, reconfiguring I/O devices, and managing large programming projects can all be greatly simplified. And, as the module library grows, increasingly sophisticated programs can be implemented using less and less new code.

This paper can be fairly termed a classic, in that it introduced the whole notion of structured analysis and structured design, as well as the concepts of coupling and cohesion, and the use of structure charts to describe the hierarchical form of a program.

Most of the material is now available in many of the books on design, which also include the subsequent revisions to the basic thinking that was introduced in this paper. The original paper is now of limited value to the teacher, and probably of even less value to the student.

Wirth71

Wirth, N. "Program Development by Stepwise Refinement." *Comm. ACM 14*, 4 (April 1971), 221-227.

Abstract: The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

An introduction to the idea of design as a series of refinements, based upon the ideas of top-down design.

The example used in the paper is the classical (and rather complex) eight queen's chess problem. While no longer providing a major source of material for the teacher, this may be a useful paper to discuss in tutorials.

Yau86

Yau, S. S., and J. J.-P. Tsai. "A Survey of Software Design Techniques." *IEEE Trans. Software Eng. SE-12*, 6 (June 1986), 713-721.

Abstract: Software design is the process which translates the requirements into a detailed design representation of a software system. Good software design is a key to produce reliable and understandable software. To support software design, many techniques and tools have been developed. In this paper, important techniques for software design, including architectural and detailed design stages, are surveyed. Recent advances in distributed software system design methodologies are also reviewed. To ensure software quality, various design verification and validation techniques are also discussed. In addition, current software metrics and error-resistant software design methodologies are considered. Future research in software design is also discussed.

This is something of a review paper, and it presents a suitably long list of references at the end, although it still manages to ignore the development of the JSD method entirely.

May provide a useful overview for the student, although it is rather concise for this purpose. A fairly comprehensive source of references for student and teacher.

. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE, MA 01731
. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) ESD/ AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003
. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 63752F; PROJECT NO. N/A; TASK NO. N/A; WORK UNIT NO. N/A
. TITLE (Include Security Classification) Introduction to Software Design	
ERSONAL AUTHOR(S) David Budgen, University of Stirling	
a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) January, 1989 15. PAGE COUNT 32
. SUPPLEMENTARY NOTATION	

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
IELD	GROUP	SUB GR.	software design design method

ABSTRACT (Continue on reverse if necessary and identify by block number)

This curriculum module provides an introduction to the principles and concepts relevant to the design of large programs and systems. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

DISTRIBUTION/AVAILABILITY OF ABSTRACT CLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ OTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c. OFFICE SYMBOL ESD/AVS (SEI JPO)

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials* package (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials* package (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

- CM-1 [superseded by CM-19]
- CM-2 Introduction to Software Design
- CM-3 The Software Technical Review Process*
- CM-4 Software Configuration Management*
- CM-5 Information Protection
- CM-6 Software Safety
- CM-7 Assurance of Software Quality
- CM-8 Formal Specification of Software*
- CM-9 Unit Testing and Analysis
- CM-10 Models of Software Evolution: Life Cycle and Process
- CM-11 Software Specifications: A Framework
- CM-12 Software Metrics
- CM-13 Introduction to Software Verification and Validation
- CM-14 Intellectual Property Protection for Software
- CM-15 Software Development and Licensing Contracts
- CM-16 Software Development Using VDM
- CM-17 User Interface Development*
- CM-18 [superseded by CM-23]
- CM-19 Software Requirements
- CM-20 Formal Verification of Programs
- CM-21 Software Project Management
- CM-22 Software Design Methods for Real-Time Systems*
- CM-23 Technical Writing for Software Engineers
- CM-24 Concepts of Concurrent Programming
- CM-25 Language and System Support for Concurrent Programming*
- CM-26 Understanding Program Dependencies

Educational Materials

- EM-1 Software Maintenance Exercises for a Software Engineering Project Course
- EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education
- EM-3 Reading Computer Programs: Instructor's Guide and Exercises